PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL

Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Perez, Pablo Ascariz

National Institute of Aerospace, Georgia Institute of Technology

SRI International, ONERA, University of A Coruña

April, 2012

Outline

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

In order to solve these two challenges this work proposes

1. Axiomatization of Lyapunov-based stability as C code annotations,
2. Implementation of linear algebra and control theory results in PVS.

In order to solve these two challenges this work proposes

1. Axiomatization of Lyapunov-based stability as C code annotations,
2. Implementation of linear algebra and control theory results in PVS.

Stability and Correctness

- The basic module for the description of a controller can be presented as

$$\xi(k+1) = f(\xi(k), \nu(k)),\ \xi(0) = \xi_0$$
$$\zeta(k) = g(\xi(k), \nu(k))$$

  where $\xi \in \mathbb{R}^n$ is the state of the controller, ν is the input of the controller and ζ is the output of the controller.
- This system is bounded-input, bounded state stable if for every ϵ there exists a δ such that $||\nu(k)|| \leq \epsilon$ implies $||\xi(k)|| \leq \delta$, for every positive integer k.

Stability and Correctness

- The basic module for the description of a controller can be presented as

$$\xi(k+1) = f(\xi(k), \nu(k)),\ \xi(0) = \xi_0$$
$$\zeta(k) = g(\xi(k), \nu(k))$$

  where $\xi \in \mathbb{R}^n$ is the state of the controller, ν is the input of the controller and ζ is the output of the controller.
- This system is bounded-input, bounded state stable if for every ϵ there exists a δ such that $||\nu(k)|| \leq \epsilon$ implies $||\xi(k)|| \leq \delta$, for every positive integer k.

- If there exists a positive definite function V such that $V(\xi(k)) \leq 1$ implies $V(\xi(k+1)) \leq 1$ then this function can be used to establish the stability of the system.
- This Lyapunov function, V, defines the ellipsoid $\{\xi | \, V(\xi) \leq 1\}$, this ellipsoid plays an important role for the stability preservation at the code level.

- If there exists a positive definite function V such that $V(\xi(k)) \leq 1$ implies $V(\xi(k+1)) \leq 1$ then this function can be used to establish the stability of the system.
- This Lyapunov function, V, defines the ellipsoid $\{\xi|\ V(\xi) \leq 1\}$, this ellipsoid plays an important role for the stability preservation at the code level.

Control software and Hoare triples

Annotated with assertions in the Hoare style we get

- $$\begin{aligned} & \{pre1\} \\ u \quad = \quad & C_c\mathbf{x}_c + D_c y_c \\ & \{post1\} \end{aligned}$$

- $$\begin{aligned} & \{pre2\} \\ \mathbf{x}_c \quad = \quad & A_c\mathbf{x}_c + B_c y_c \\ & \{post2\}. \end{aligned}$$

An ellipsoid-aware Hoare logic

- To use ellipsoids to formally specify bounded input, bounded state stability in.
- Typically, an instruction S would be annotated in the following way:

$$\{x \in \mathcal{E}_P\}\ y = Ax + b\ \{y - b \in \mathcal{E}_Q\} \tag{1}$$

where the pre- and post- conditions are predicates expressing that the variables belong to some ellipsoid, with $\mathcal{E}_p = \{x : \mathbb{R}^n | x^T P^{-1} x \leq 1\}$ and $Q = APA^T$.

An ellipsoid-aware Hoare logic

- To use ellipsoids to formally specify bounded input, bounded state stability in.
- Typically, an instruction S would be annotated in the following way:

$$\{x \in \mathcal{E}_P\}\ y = Ax + b\ \{y - b \in \mathcal{E}_Q\} \tag{1}$$

where the pre- and post- conditions are predicates expressing that the variables belong to some ellipsoid, with $\mathcal{E}_p = \{x : \mathbb{R}^n | x^T P^{-1} x \leq 1\}$ and $Q = APA^T$.

An ellipsoid-aware Hoare logic

The mathematical theorem that guarantees the relations is :

Theorem

If M, Q are invertible matrices, and
$(x-c)^T Q^{-1}(x-c) \leq 1$ *and*
$y = Mx + b$
then
$(y-b-Mc)^T(MQM^T)^{-1}(y-b-Mc) \leq 1$

We will refer to it as the *ellipsoid affine combination theorem*.

Verification conditions

A Matlab program

```
1:  A = [0.4990, -0.0500;
         0.0100, 1.0000];
2:  C = [-564.48, 0];
3:  B = [1;0];D = 1280;
4:  x = zeros(2,1);
5:  while 1
6:    y = fscanf(stdin,"%f");
7:    y = max(min(y,1),-1);
8:    u = C*x + D*y;
9:    fprintf(stdout,"%f\n",u);
10:   x = A*x + B*y;
11: end
```

$\{true\}$
`4:  x = zeros(2,1)`
$\{\boldsymbol{x} \in \mathcal{E}_P\}$.
`5:  while 1`
$\{x \in \mathcal{E}_P\}$
`6:  y = fscanf(stdin,"%f")`
$\{\boldsymbol{x} \in \mathcal{E}_P\}$
`7:  y = max(min(y,1),-1);`
$\{\boldsymbol{x} \in \mathcal{E}_P,\ \boldsymbol{y}^2 \leq 1\}$
`8:  u = C*x+D*y;`
$\{\boldsymbol{x} \in \mathcal{E}_P,\ \boldsymbol{u}^2 \leq 2(\boldsymbol{C}\boldsymbol{P}^{-1}\boldsymbol{C}^T + \boldsymbol{D}^2),\ \boldsymbol{y}^2 \leq 1\}$
`9:  fprintf(stdout,"%f\n",u)`
$\{x \in \mathcal{E}_P,\ y^2 \leq 1\}$

?

$\{Ax + By \in \mathcal{E}_P,\ y^2 \leq 1,$
$u^2 \leq 2(CP^{-1}C^T + D^2)\}$
`9:  fprintf(stdout,"%f\n",u)`
$\{\boldsymbol{Ax} + \boldsymbol{By} \in \mathcal{E}_P,\ \boldsymbol{y}^2 \leq 1\}$
`10:  x = A*x + B*y;`
$\{\boldsymbol{x} \in \mathcal{E}_P\}$
`11: end`
$\{false\}$

- Now that we know the annotations that we want to generate on the code, we have to find a concrete way to express them on actual C code.
- The ANSI/ISO C Specification Language (ACSL) allows its user to specify the properties of a C program within comments,
- This language was proposed as part of the Frama-C platform,

- Now that we know the annotations that we want to generate on the code, we have to find a concrete way to express them on actual C code.
- The ANSI/ISO C Specification Language (ACSL) allows its user to specify the properties of a C program within comments,
- This language was proposed as part of the Frama-C platform,

- Now that we know the annotations that we want to generate on the code, we have to find a concrete way to express them on actual C code.
- The ANSI/ISO C Specification Language (ACSL) allows its user to specify the properties of a C program within comments,
- This language was proposed as part of the Frama-C platform,

Verification conditions

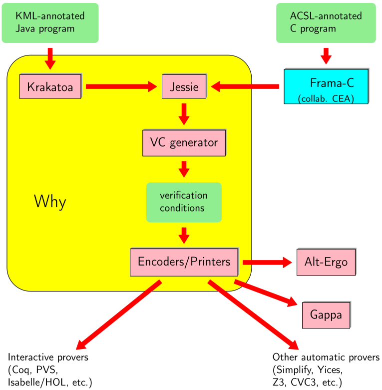

Figure 2: Verification

- We outline the axiomatization in ACSL to fit our needs, which consist of expressing ellipsoid-based Hoare triples over C code.
- We first present the axiomatization of linear algebra elements in ACSL.
- Then we present the Hoare triple annotations in ACSL and the POs generated by them.

- We outline the axiomatization in ACSL to fit our needs, which consist of expressing ellipsoid-based Hoare triples over C code.
- We first present the axiomatization of linear algebra elements in ACSL.
- Then we present the Hoare triple annotations in ACSL and the POs generated by them.

- We outline the axiomatization in ACSL to fit our needs, which consist of expressing ellipsoid-based Hoare triples over C code.
- We first present the axiomatization of linear algebra elements in ACSL.
- Then we present the Hoare triple annotations in ACSL and the POs generated by them.

- The following abstract types are declared:

ACSL
```
//@ type matrix; type vector
```

- With these abstract types, basic matrix operations and properties are introduced

ACSL
```
@ logic real mat_select(matrix A, integer i, integer j);
@ logic integer mat_row(matrix A);
@ logic integer mat_col(matrix A);
```

- The following abstract types are declared:

ACSL
```
//@ type matrix; type vector
```

- With these abstract types, basic matrix operations and properties are introduced

ACSL
```
@ logic real mat_select(matrix A, integer i, integer j);
@ logic integer mat_row(matrix A);
@ logic integer mat_col(matrix A);
```

- The multiplication of a matrix with a vector is defined with function vect_mult($\mathtt{matrix}\ A,\ \mathtt{vector}\ x$), which returns a vector.
- Addition and multiplication of 2 matrices, multiplication by a scalar, and inverse of a matrix are declared as matrix types

- The multiplication of a matrix with a vector is defined with function vect_mult($\mathsf{matrix}\ A,\ \mathsf{vector}\ x$), which returns a vector.
- Addition and multiplication of 2 matrices, multiplication by a scalar, and inverse of a matrix are declared as matrix types

inverse of a matrix A, mat_inverse(A) is defined using the predicate is_invertible(A) as follows:

ACSL

```
/*@ axiom mat_inv_select_i_eq_j:
@ ∀matrixA, integer i, j;
@ is_invertible(A) && i == j ==>
@ mat_select(mat_mult(A, mat_inverse(A)), i, j) = 1
@
@ axiom mat_inv_select_i_dff_j:
@ ∀matrixA, integer i, j;
@ is_invertible(A) && i! = j ==>
@ mat_select(mat_mult(A, mat_inverse(A)), i, j) = 0
@*/
```

Complex constructions or relations can be defined as uninterpreted predicates. The following predicate is meant to express that vector x belongs to $\mathcal{E}_{\mathcal{P}}$:

- ACSL
  ```
  //@ predicate in_ellipsoid(matrix P, vector x);
  ```
- mat_of_array or vect_of_array, is used to associate an ACSL matrix type to a C array.
  ACSL
  ```
  //@ logic matrix mat_of_array{L}(float *A, integer row,
  integer col);
  ```

Complex constructions or relations can be defined as uninterpreted predicates. The following predicate is meant to express that vector x belongs to $\mathcal{E}_{\mathcal{P}}$:

- ACSL
  ```
  //@ predicate in_ellipsoid(matrix P, vector x);
  ```

- mat_of_array or vect_of_array, is used to associate an ACSL matrix type to a C array.

  ACSL
  ```
  //@ logic matrix mat_of_array{L}(float *A, integer row,
  integer col);
  ```

ACSL

```
// @ axiom mat_of_array_select:
@ forall float *A; forall integer i, j, k, l;
@ mat_select(mat_of_array(A, k, l), i, j) == A[l*i+j];
```

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ require P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ require P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ require P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ require P
  //@ ensures R
  Q

- We need to deal with memory issues. In general, we want all functions to be called with valid pointers as arguments, i.e., valid array and therefore valid matrices.
- This is what the built-in ACSL predicate valid does. The followings snippet shows how the contract can be written using mat_select and mat_of_array,

ACSL

```
/*@ requires (valid(a + (0..3)));
@ ensures \forall integer i,j; 0 <= i < 2 && 0 <= j < 2
@ ==> mat_select(mat_of_array(a, 2, 2), i, j) == 0;
@ */
void zeros_2x2(float* a)
{ a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

- We need to deal with memory issues. In general, we want all functions to be called with valid pointers as arguments, i.e., valid array and therefore valid matrices.
- This is what the built-in ACSL predicate valid does. The followings snippet shows how the contract can be written using mat_select and mat_of_array,

ACSL

```
/*@ requires (valid(a + (0..3)));
@ ensures \forall integer i,j; 0 <= i < 2 && 0 <= j < 2
@ ==> mat_select(mat_of_array(a,2,2),i,j) == 0;
@ */
void zeros_2x2(float* a)
{ a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

- We need to deal with memory issues. In general, we want all functions to be called with valid pointers as arguments, i.e., valid array and therefore valid matrices.
- This is what the built-in ACSL predicate valid does. The followings snippet shows how the contract can be written using mat_select and mat_of_array,

ACSL

```
/*@ requires (valid(a + (0..3)));
@ ensures \forall integer i,j; 0 <= i < 2 && 0 <= j < 2
@ ==> mat_select(mat_of_array(a, 2, 2), i, j) == 0;
@ */
void zeros_2x2(float* a)
{ a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

ACSL

```
@ Ac = mat_of_2x2_scalar(0.449,-0.05,0.01,1.);
@ P = mat_of_2x2_scalar(1.5325,10.0383,10.0383,507.2450);
@ Q = mat_mult(mat_inv(transpose(Ac)),mat_mult(P,mat_inv(Ac)));
*/
@ requires (valid(xc + (0..1)));
@ requires (valid(yc + (0..1)));
@ requires in_ellipsoid(P,vect_of_array(xc,2));
@ ensures in_ellipsoid(Q,vect_of_array(yc,2));*/
void inst2(float* xc, float* yc)
{ yc[0]= 0.449*xc[0] + -0.05*xc[1];
yc[1]= .01*xc[0] + 1.*xc[1]; }
```

- Errors due to floating point approximations are thus not taken into account.
- The Frama-C toolset offers the possibility of making this assumption by including 'JessieFloatModel(Math)'.

- Errors due to floating point approximations are thus not taken into account.
- The Frama-C toolset offers the possibility of making this assumption by including 'JessieFloatModel(Math)'.

- Frama-C tools do not require an annotation at each line as proposed by Hoare.
- They rather rely weakest precondition calculus.
- The proof obligation (PO) is then $P \implies wp(S, Q)$ where P is the pre-condition.

- Frama-C tools do not require an annotation at each line as proposed by Hoare.
- They rather rely weakest precondition calculus.
- The proof obligation (PO) is then $P \implies wp(S,Q)$ where P is the pre-condition.

- Frama-C tools do not require an annotation at each line as proposed by Hoare.
- They rather rely weakest precondition calculus.
- The proof obligation (PO) is then $P \implies wp(S, Q)$ where P is the pre-condition.

- On the one hand, using ACSL and the Frama-C framework, we were able to generate POs about the ellipsoid predicate.
- Frama-C tools even make it possible to express the PO in PVS, along with a complete axiomatisation in PVS of C programs semantics.

- On the one hand, using ACSL and the Frama-C framework, we were able to generate POs about the ellipsoid predicate.
- Frama-C tools even make it possible to express the PO in PVS, along with a complete axiomatisation in PVS of C programs semantics.

PVS

```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```

PVS

```
vect_of_array(yc, 2, floatP_floatM0)'vect =
Ac * vect_of_array(xc, 2, floatP_floatM)'vect
```

For both POs,

- we must first interpret the uninterpreted types and to prove the properties that are defined axiomatically.
- We must then discharge the verification conditions. This is done by using PVS and a linear algebra extension of it.

PVS

```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```

PVS

```
vect_of_array(yc, 2, floatP_floatM0)'vect =
Ac * vect_of_array(xc, 2, floatP_floatM)'vect
```

For both POs,

- we must first interpret the uninterpreted types and to prove the properties that are defined axiomatically.
- We must then discharge the verification conditions. This is done by using PVS and a linear algebra extension of it.

- PVS
  ```
  Mapping:TYPE= [# dom: posnat, codom: posnat, mp:
  [Vector[dom]->Vector[codom]] #]
  ```
- PVS
  L(n,m)(f) = (# rows:=m, cols:=n, matrix:=λ(j,i): f`mp(e(n)(i))(j) #)
  T(n,m)(A) = (# dom:=n, codom:=m, mp:=λ(x,j): $\Sigma_{i=0}^{A`cols-1}$($\lambda$(i): A`matrix(j,i)*x(i) #))
- PVS
  Matrix_inv(n):TYPE = {A: Square | squareMat?(n)(A) and bijective?(n)(T(n,n)(A))}
- PVS
  inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))

PVS

- `Mapping:TYPE= [# dom: posnat, codom: posnat, mp: [Vector[dom]->Vector[codom]] #]`

PVS

- L(n,m)(f) = (# rows:=m, cols:=n, matrix:=λ(j,i): f‘mp(e(n)(i))(j) #)
  T(n,m)(A) = (# dom:=n, codom:=m, mp:=λ(x,j): $\Sigma_{i=0}^{A‘cols-1}$(λ(i): A‘matrix(j,i)*x(i) #))

PVS

- Matrix_inv(n):TYPE = {A: Square | squareMat?(n)(A) and bijective?(n)(T(n,n)(A))}

PVS

- `inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))`

- PVS
  ```
  Mapping:TYPE= [# dom: posnat, codom: posnat, mp:
  [Vector[dom]->Vector[codom]] #]
  ```
- PVS
  L(n,m)(f) = (# rows:=m, cols:=n, matrix:=λ(j,i): f`mp(e(n)(i))(j) #)
  T(n,m)(A) = (# dom:=n, codom:=m, mp:=λ(x,j): $\Sigma_{i=0}^{A\text{`cols}-1}$($\lambda$(i): A`matrix(j,i)*x(i) #))
- PVS
  Matrix_inv(n):TYPE = $\{$A: Square | squareMat?(n)(A) and bijective?(n)(T(n,n)(A))$\}$
- PVS
  ```
  inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))
  ```

- PVS
```
Mapping:TYPE= [# dom: posnat, codom: posnat, mp:
[Vector[dom]->Vector[codom]] #]
```

- PVS
```
L(n,m)(f) = (# rows:=m, cols:=n, matrix:=λ(j,i):
f`mp(e(n)(i))(j) #)
T(n,m)(A) = (# dom:=n, codom:=m, mp:=λ(x,j): Σ_{i=0}^{A`cols−1}(λ(i):
A`matrix(j,i)*x(i) #))
```

- PVS
```
Matrix_inv(n):TYPE = {A: Square | squareMat?(n)(A) and
bijective?(n)(T(n,n)(A))}
```

- PVS
```
inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))
```

PVS

```
ellipsoid_affine_comb: LEMMA ∀ (n:posnat, Q, M: SquareMat(n), x,
y, b, c: Vector[n]):
bijective?(n)(T(n,n)(Q)) AND bijective?(n)(T(n,n)(M))
AND (x-c)*(inv(n)(Q)*(x-c))≤ 1
AND y=M*x + b
IMPLIES
(y-b-M*c)*(inv(n)(M*(Q*transpose(M)))*(y-b-M*c))≤ 1
```

- We have developed a PVS library that is able to reason about these properties.
- We now must link these two worlds: ACSL ellipsoids predicate proof obligation in PVS must be connected with with our linear algebra PVS library.

- We have developed a PVS library that is able to reason about these properties.
- We now must link these two worlds: ACSL ellipsoids predicate proof obligation in PVS must be connected with with our linear algebra PVS library.

Verification conditions and theory interpretation

- Theory interpretation is a logical technique for relating one axiomatic theory to another.
- Interpretations can be used to show:
- An axiomatically defined specification is consistent
- or that a axiomatically defined specification captures its intended models.

Verification conditions and theory interpretation

- Theory interpretation is a logical technique for relating one axiomatic theory to another.
- Interpretations can be used to show:
- An axiomatically defined specification is consistent
- or that a axiomatically defined specification captures its intended models.

Verification conditions and theory interpretation

- Theory interpretation is a logical technique for relating one axiomatic theory to another.
- Interpretations can be used to show:
- An axiomatically defined specification is consistent
- or that a axiomatically defined specification captures its intended models.

PVS

```
IMPORTING acsl_theory{{ matrix := Matrix,
vector := Vector_no_param,
vect_length := LAMBDA (v:Vector_no_param): v'length,
mat_row := LAMBDA (M:Matrix): M'rows,
mat_col := LAMBDA (M:Matrix): M'cols,
mat_mult := *,
in_ellipsoid := in_ellipsoid?
mat_inv := LAMBDA (M:Matrix): IF square?(M) THEN IF
bijective?(M'rows)(T(M'rows,M'rows)(M))
THEN inv(M'rows)(M)
ELSE M
ENDIF
ELSE M ENDIF }}
```

PVS

```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```

PVS

```
bijections :LEMMA
bijective?(2)(T(2,2)(P_0)) AND  bijective?(2)(T(2,2)(Ac))
```

where $Ac = mat\_of\_2x2\_scalar(0.449, -0.05, 0.01, 1.)$ and $P = mat\_of\_2x2\_scalar(1.5325, 10.0383, 10.0383, 507.2450)$

- PVS
```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```
- PVS
```
bijections :LEMMA
bijective?(2)(T(2,2)(P_0)) AND  bijective?(2)(T(2,2)(Ac))
```

where $Ac = mat\_of\_2x2\_scalar(0.449, -0.05, 0.01, 1.)$ and $P = mat\_of\_2x2\_scalar(1.5325, 10.0383, 10.0383, 507.2450)$

Conclusions

- We have described a global approach to validate stability properties of C code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- proving the stability of the control code using ellipsoid affine combinations.
- We have defined an ACSL extension to describe predicates over the code, as well as a PVS library able to manipulate these predicates.

Conclusions

- We have described a global approach to validate stability properties of C code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- proving the stability of the control code using ellipsoid affine combinations.
- We have defined an ACSL extension to describe predicates over the code, as well as a PVS library able to manipulate these predicates.

Conclusions

- We have described a global approach to validate stability properties of C code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- proving the stability of the control code using ellipsoid affine combinations.
- We have defined an ACSL extension to describe predicates over the code, as well as a PVS library able to manipulate these predicates.

Conclusions

- We have described a global approach to validate stability properties of C code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- proving the stability of the control code using ellipsoid affine combinations.
- We have defined an ACSL extension to describe predicates over the code, as well as a PVS library able to manipulate these predicates.

- Theory interpretation maps proof obligations generated from the code to their equivalent in this PVS library.
- This mapping allows to discharge POs using the ellipsoid affine combination theorem implemented in PVS.
- Linear algebra PVS libraries can be used for the formal specification of control theory properties

- Theory interpretation maps proof obligations generated from the code to their equivalent in this PVS library.
- This mapping allows to discharge POs using the ellipsoid affine combination theorem implemented in PVS.
- Linear algebra PVS libraries can be used for the formal specification of control theory properties

- Theory interpretation maps proof obligations generated from the code to their equivalent in this PVS library.
- This mapping allows to discharge POs using the ellipsoid affine combination theorem implemented in PVS.
- Linear algebra PVS libraries can be used for the formal specification of control theory properties

- Theory interpretation maps proof obligations generated from the code to their equivalent in this PVS library.
- This mapping allows to discharge POs using the ellipsoid affine combination theorem implemented in PVS.
- Linear algebra PVS libraries can be used for the formal specification of control theory properties

The authors would like to thank

- A. Goodloe for his suggestion of the use of the Frama-C toolset and his help in axiomatising of linear algebra in ACSL.